Software Synthesis with ACL2

Eric Smith
Kestrel Institute

ACL2 Workshop 2015

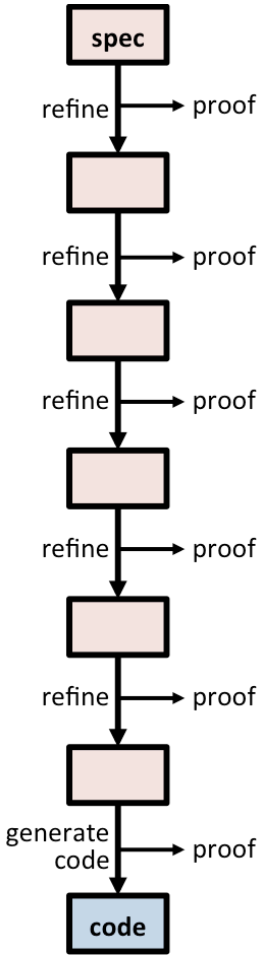
Outline

- Overview of Refinement-Based Synthesis
- Proof-Emitting Transformations
- Specs and Morphisms

- Joint work with Alessandro Coglio and others.

Refinement-Based Synthesis

- Derive an implementation from a specification via a sequence of provably correct refinement steps.
  - Each step represents a design decision
- Not a new idea, e.g.,
  - Z notation
  - B method
  - VDM (Vienna Development Method)
  - Specware
  - defspec / defrefine / …
- New implementation in ACL2



Part 1: Proof-Producing Transformations

- make-tail-rec
  - assoc
  - monoid
  - nats counting down
  - bit vectors counting down
- finite-difference
- remove-cdring
- expand-lets
- letify
- simplify-body
- rewrite-body
- drop-function-from-nest
- drop-irrelevant-params
- flatten-params
- homogenize-tail-rec
- flip-if
- rename-params
- reorder-params
- restructure-elseif
- make-do-while
- lift-condition
- weaken
- strengthen
- wrap-output
- wrap-input
- undo-finite-difference
- undo-make-tail-rec
- define-op
- copy-spec
- spec substitution
- decrease-and-conquer
- divide-and-conquer

Make-tail-rec Transformation

```
(defun sum-squares (n)
  (if (zp n)
      0
    (+ (* n n) (sum-squares (- n 1)))))
```

```
(make-tail-rec-monoid sum-squares :domain natp)
```

```
(DEFUN SUM-SQUARES$1 (N ACC)
  (IF (ZP N)
      ACC
    (SUM-SQUARES$1 (- N 1) (+ ACC (* N N)))))

(DEFUN SUM-SQUARES$1-WRAPPER (N)
   (SUM-SQUARES$1 N 0))

(DEFTHM SUM-SQUARES-IS-SUM-SQUARES$1-WRAPPER
   (EQUAL (SUM-SQUARES N)
          (SUM-SQUARES$1-WRAPPER N)))
```

Finite-differencing Transformation

```
(DEFUN SUM-SQUARES$1 (N ACC)
  (IF (ZP N)
      ACC
    (SUM-SQUARES$1 (- N 1) (+ ACC (* N N)))))
```

```
(finite-difference sum-squares$1 (* n n) *math-rules*)
```

```
(DEFUN SUM-SQUARES$2 (N ACC N*N)
   (IF (ZP N)
       ACC
     (SUM-SQUARES$2 (- N 1)
                    (+ ACC N*N)
                    (+ N*N (- N) (- N) 1))))

(DEFUN SUM-SQUARES$2-WRAPPER (N ACC)
        (SUM-SQUARES$2 N ACC (* N N)))

(DEFTHM SUM-SQUARES$1-BECOMES-SUM-SQUARES$2
  (EQUAL (SUM-SQUARES$1 N ACC)
         (SUM-SQUARES$2-WRAPPER N ACC))
```

```
Maintain invariant for n*n:
  (* (- n 1) (- n 1)
= (+ (* n n) (- n) (- n) (* -1 -1))
= (+ (* n n) (- n) (- n) 1)
= (+ n*n     (- n) (- n) 1)
```

Remove-cdring Transformation

```
(DEFUN MEMBERP (A X)
  (IF (CONSP X)
      (IF (EQUAL A (CAR X))
          T
        (MEMBERP A (CDR X)))
    NIL))
```

(remove-cdring memberp)

```
(DEFUN MEMBERP$1 (A X N)
  (IF (NOT (NATP N)) ; should never happen
      T
    (IF (CONSP (NTHCDR N X))
        (IF (EQUAL A (CAR (NTHCDR N X)))
            T
          (MEMBERP$1 A X (+ 1 N)))
      NIL)))

(DEFUN MEMBERP$1-WRAPPER (A X)
  (MEMBERP$1 A X 0))

(DEFTHM MEMBERP-BECOMES-MEMBERP$1-WRAPPER
  (EQUAL (MEMBERP A X)
         (MEMBERP$1-WRAPPER A X)))
```

Implementation of Transformations

- Use macros / make-event
  - Look up the target function in the logical world
  - Generate new function(s)
  - Generate correctness theorems
- Carefully control the proofs
- All transformations prove correctness of their output
  - "Verifying compiler" approach

Part 2: Specs and Morphisms

- Specware-style refinement in ACL2

Specs

- Spec: a logical theory
  - collection of 'ops' (functions), constraints (axioms), theorems, and imports of other specs

- Example:

```
(spec sorting
  (op my-sort (l))
  (axiom my-sort-correct
    (implies (true-listp l)
             (sorted-permutationp (my-sort l) l))))
```

Morphisms

- Morphism: a theory interpretation
  - i.e., a property-preserving mapping between specs.
  - Maps names to names
  - Proof obligation: Axioms in source spec must be theorems in target spec

Trivial Example

```
(spec s1
  (op foo (x))
  (op bar (x))
  (axiom foo-bar
     (<= (foo x) (bar x))))
```

```
(morphism m
  (s1 s2)
  ((foo foo2)
   (bar bar2)))
```

Proof Obligation:

```
(<= (foo2 x) (bar2 x))
```

```
(spec s2
  (op foo2 (x))
  (axiom natp-of-foo2
    (natp (foo2 x)))
  (op bar2 (x)
    (* 2 (foo2 x))))
```

ACL2 Implementation

- Uses macros and make-event
  - Track everything with tables
- Spec: basically an encapsulate
  - Specs can import other specs (allows hierarchy)
  - No witness needed for consistency (next slide…)
  - But still get a conservative extension of the ACL2 world
- Morphism: basically the obligations for a functional instantiation

Avoiding Giving Witnesses

- Encapsulate requires witnesses for the constrained functions
  - Ensures the constraints are satisfiable
  - Amounts to writing and proving an implementation
    - But this is what refinement will ultimately do!
- Specs don't require witnesses
  - Every spec with axioms gets a 0-ary "witness predicate"
    - Added as a constrained function in the encapsulate
    - Locally defined to be false
    - Added as an assumption for every axiom and theorem
  - Morphism maps source witness predicate to target witness predicate
  - A fully-defined spec (last refinement step) doesn't have a witness predicate
    - The final morphism maps the witness predicate to true.

Spec Substitution

- Spec Substitution: Specialize/Concretize a spec by applying a morphism
  - Related to a categorical “push-out”

- Implementation: Basically a functional instantiation

Divide and Conquer

```
(spec divide-and-conquer-problem
 (op problemp (x))
 (op solutionp (solution problem))
 (op directly-solvablep (problem))
 (op solve-directly (problem))
 (op divide (problem) :output (mv * *))
 (op combine (solution1 solution2))

 ...

 (axiom solve-directly-correct
   (implies (and (problemp problem)
                 (directly-solvablep problem))
            (solutionp (solve-directly problem) problem)))

 (axiom combine-correct
   (implies (and (problemp problem)
                 (not (directly-solvablep problem))
                 (solutionp solution0 (mv-nth '0 (divide problem)))
                 (solutionp solution1 (mv-nth '1 (divide problem))))
            (solutionp (combine solution0 solution1) problem)))
  ...)
```

Divide and Conquer

```
(spec divide-and-conquer-problem
 (op problemp (x))
 (op solutionp (solution problem))
 (op directly-solvablep (problem))
 (op solve-directly (problem))
 (op di
 (op co
 ...
 (axiom
   (imp
 (axiom
   (imp
 ...)
```

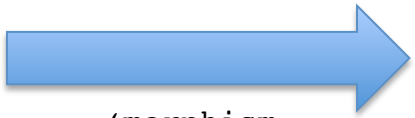
```
(spec divide-and-conquer-solution
  (import divide-and-conquer-problem)

  (op solve (problem)
    (if (not (problemp problem))
        :default-value ;should never happen.
      (if (directly-solvablep problem)
          (solve-directly problem)
        (mv-let (first-subproblem second-subproblem)
                (divide problem)
                (combine (solve first-subproblem)
                         (solve second-subproblem))))))

  (theorem solve-correct
    (implies (problemp problem)
             (solutionp (solve problem) problem))))
```

Merge Sort

```
(spec merge-sort-problem

  ;;Returns (mv part1 part2)
  (op split-list (lst)
    (if (endp lst)
        (mv nil nil)
      (if (endp (rest lst))
          (mv (list (first lst)) nil)
        (mv-let (part1 part2)
                (split-list (rest (rest lst)))
                (mv (cons (first lst) part1)
                    (cons (second lst) part2))))))

  ;; Merge the sorted lists LST1 and LST2 into a sorted list.
  (op merge-lists (lst1 lst2)
    (declare (xargs :measure (+ (len lst1) (len lst2))))
    (if (endp lst1)
        lst2
      (if (endp lst2)
          lst1
        (if (< (first lst1) (first lst2))
            (cons (first lst1) (merge-lists (rest lst1) lst2))
          (cons (first lst2) (merge-lists lst1 (rest lst2)))))))

  ;;Recognizes directly solvable instances:
  (op short-listp (lst) (< (len lst) 2))

  ;; A list of length 0 or 1 is already sorted:
  (op do-nothing (lst) lst)

  ...)
```

Applying Divide and Conquer Step 1 (prove the morphism)

```
(spec divide-and-conquer-problem
 (op divide …)
 (op combine …)
 (axiom …)
  …)
```



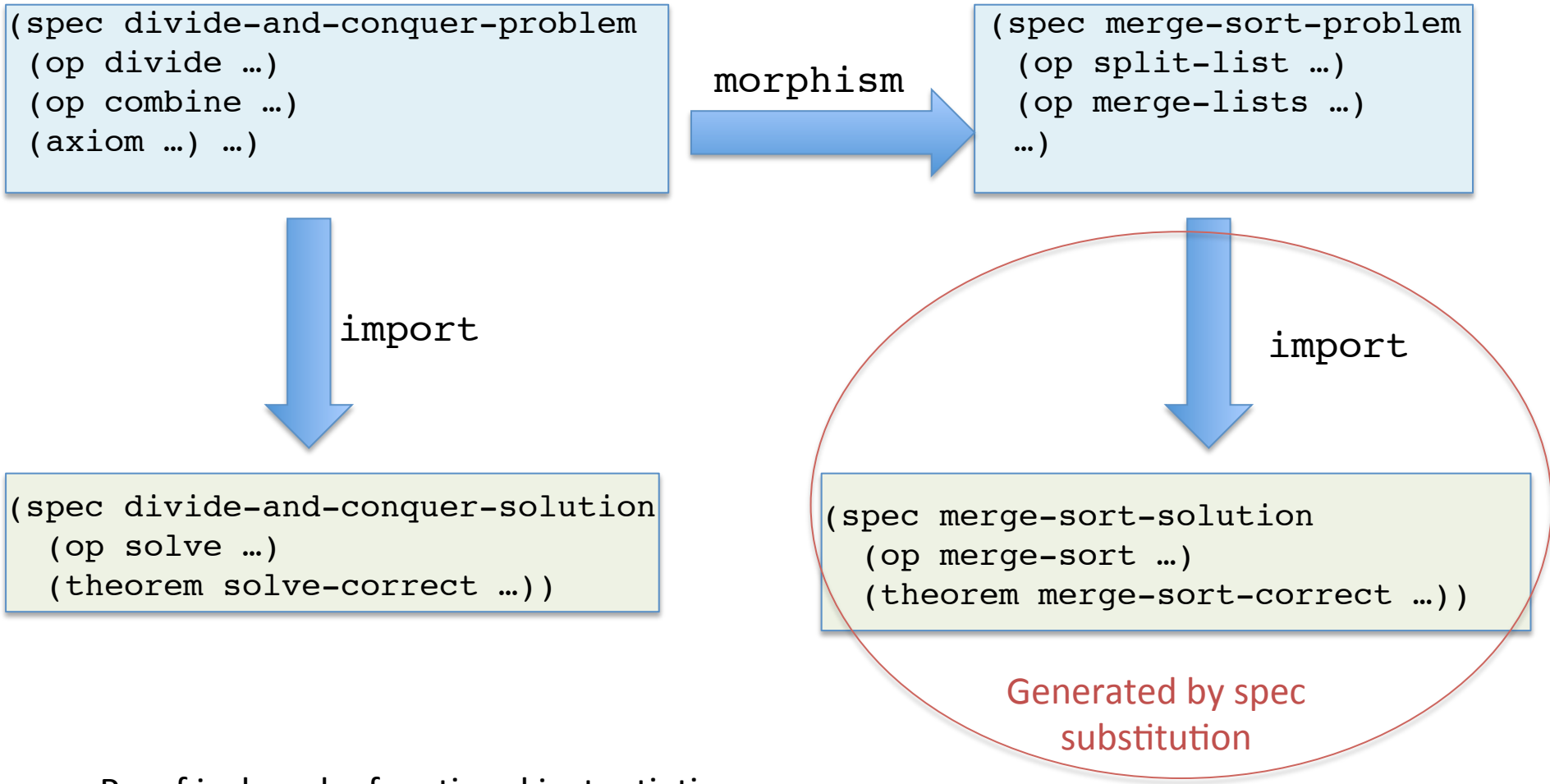
```
(spec merge-sort-problem
  (op split-list …)
  (op merge-lists …)
  …)
```

```
(morphism …
((problemp true-listp)
 (solutionp sorted-permutationp)
 (directly-solvablep short-listp)
 (solve-directly do-nothing)
 (divide split-list)
 (combine merge-lists) …)
```

Proof Obligations:

```
(theorem merge-sort-solve-directly-correct
   (implies (and (true-listp lst)
                 (short-listp lst))
            (sorted-permutationp (do-nothing lst) lst)))

(theorem merge-lists-correct
   (implies (and (true-listp problem)
                 (not (short-listp problem))
                 (sorted-permutationp solution0
                                      (mv-nth 0 (split-list problem)))
                 (sorted-permutationp solution1
                                      (mv-nth 1 (split-list problem))))
            (sorted-permutationp (merge-lists solution0 solution1)
                                 problem)))
...
```

Applying Divide and Conquer Step 1 (apply Spec Substitution)



- Proof is done by functional instantiation.
- Normally, a derivation would have more than one step.

Conclusion

- Software Synthesis in ACL2
  - Stepwise Refinement
  - Transformations
  - Specs and Morphisms
- Correct-by-construction Code
- Variations:
  - Generate diverse versions
  - Verify existing code